Automatic Tuning of the Multiprogramming Level in Sybase SQL Anywhere

Mohammed Abouzour
Sybase iAnywhere Solutions
mabouzou@sybase.com

Kenneth Salem
University of Waterloo
kmsalem@uwaterloo.ca

Peter Bumbulis
Sybase iAnywhere Solutions
bumbulis@sybase.com

***Abstract*— This paper looks at the problem of automatically tuning the database server multiprogramming level to improve database server performance under varying workloads. We describe two tuning algorithms that were considered and how they performed under different workloads. We then present the hybrid approach that we have successfully implemented in SQL Anywhere 12. We found that the hybrid approach yielded better performance than each of the algorithms separately.**

I. Introduction

Database servers are used in a variety of software environments. Each application has its own workload characteristics. As a result, a database server must be configured to achieve the best performance possible for a specific workload on specific hardware.

One of the many configuration parameters that database servers have is the multiprogramming level (MPL). The MPL is the concurrency level of the server. The concurrency level of the server is measured by how many requests are allowed to execute concurrently. Setting the MPL too high or too low can have an adverse effect on performance. A high MPL setting might cause increased contention on shared resources (e.g. the buffer pool) while a low MPL might limit the concurrency level of the server.

In this paper we discuss the problem of choosing the MPL setting. We show how two algorithms, Hill Climbing and Global Parabola approximation, performed in automatically tuning the MPL settings. We then present the hybrid approach that we implemented in SQL Anywhere 12 along with challenges and issues that we have encountered. We found that the hybrid approach had better performance than each of the algorithms separately.

In the following section we give a description of the problem. In section III we provide the details of the design and implementation of two automatic tuning algorithms. Section IV and V describe our testing methodology and results. Section VI discusses our practical experience with implementing automatic tuning of the MPL setting in SQL Anywhere 12 and also lessons learned. Finally, section VII presents a summary of our findings.

II. Background and Related Work

In recent years, researchers have started looking at ways to help reduce the total cost of ownership of maintaining and operating a database system. A new area of research has emerged that aims to create what are called *self-tuning database systems* ([1], [2], [3]). Database servers with self-tuning algorithms respond to changing workload characteristics or operating system conditions with minimal or virtually no DBA intervention. Two areas in which self-tuning algorithms have been successfully used are automatic cache management ([4], [5]) and self-tuning histograms [6].

Many of the widely used commercial database servers (e.g., MS SQLServer [7], Oracle, and SQL Anywhere [8]) employ a worker-per-request server architecture. In this architecture, there is one queue that is used to queue all database server requests from all connections. A worker dequeues a request from the request queue, processes the request and then goes back to process the next available request in the queue or block on an idle queue. In this configuration, there are no guarantees that a single connection will be serviced by the same worker. This architecture has proved to be more effective in handling large numbers of connections and it has less overhead [9] (if configured properly). The difficult issue with this architecture is how to set the size of the worker pool to achieve good throughput levels [10]. This parameter effectively controls the MPL of the server. DBAs have two choices when setting the value of this parameter:

1) Large number of workers: Using a large MPL can allow the server to process a large number of requests simultaneously. However, the drawback is that there is a substantial risk that a large number of workers can put the server into a thrashing state due to hardware or software resource contention or to excessive context switching between threads ([11], [12]). Another issue with a large number of workers is that each worker has its own stack. Having many workers can consume a substantial part of a process's address space. This can negatively affect the size of the database server buffer pool that is available to cache database pages. This issue is more critical for 32-bit database servers.
2) Small number of workers: The immediate benefit of this approach is that the server has more memory to be used for caching; however, the drawback is that the MPL of the server is reduced and the hardware resources are under-utilized.

In order to decide on the optimal number of workers, DBAs could run load tests as part of their capacity planning process to find a value that can achieve the minimum required response time needed by the database application. Bowei et

al. describe a smart hill-climbing approach to configuring application servers [13]. A similar algorithm can be applied to database servers. Although the experimental approach can be very effective in configuring the database server MPL parameter, it suffers from some drawbacks. One of the drawbacks is that this parameter setting is only good under the tested conditions and might not handle different or changing workload characteristics. In addition, this discovered value is very specific to the hardware on which the experiments were performed. If a hardware upgrade is performed, the selected value might be too conservative for the new hardware.

One of the earliest papers that exposed the problem of automatic tuning of MPL in database systems was by Heiss and Wagner [14]. They outlined different sources of contention and suggested a possible feedback control mechanism that uses two different algorithms: an Incremental Steps (IS) approach and a Parabola Approximation (PA) approach. Our work extends on those algorithms. While Heiss and Wagner used simulations to validate their findings, we used a commercial database server and an industry-standard benchmark TPC-C [15].

Recent research by Bianca Schroeder and colleagues [9] used an external controller to adjust the MPL. Their work studied the affects of MPL on throughput and mean response time. Their controller used a simple feedback control loop. Our work is an extension of their work as we looked at two different algorithms that could be used by the controller.

In this paper we are proposing an online controller that can be used to automatically adjust the multiprogramming level of the database server on-the-fly to achieve maximum throughput.

III. Design and Implementation

In this section we describe the architecture of our controller and the two different algorithms that we initially considered.

A. Controller Architecture

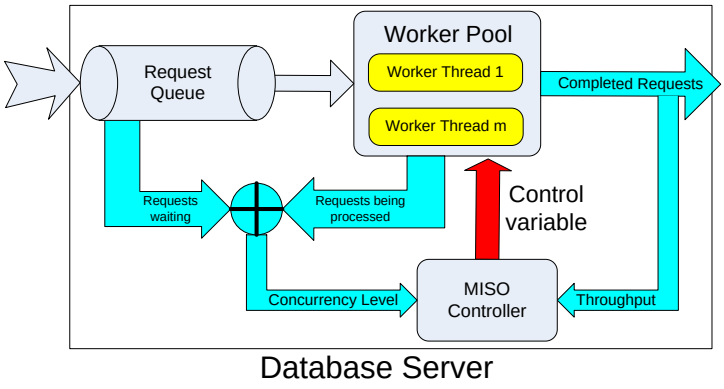


Fig. 1: Multi-Input Single-Output Controller Architecture

Any controller design involves a set of inputs and a set of outputs. We will be using a multi-input single-output (MISO) controller. Figure 1 illustrates the architecture of the controller. For inputs, our controller will monitor the throughput level of the server. This is measured by the number of server-side requests (not transactions) completed. A single database transaction can involve one or more server side requests.

The second input parameter to the controller is the total number of requests that are outstanding at the server. We will call this number the *workload concurrency level* and it is the sum of the number of requests that are waiting in the request queue and the number of requests that are being serviced by the worker pool. For example, if the worker pool size is 10 and all of the 10 workers are busy servicing requests and there are 5 requests waiting in the request queue, then the workload concurrency level is 15.

The third input parameter is the control interval. If the interval is too small, then the controller will be reacting to potentially noisy input measurements that are caused by natural fluctuations in the workload. A small interval will also not give the new control setting sufficient time to have an effect on throughput before the next input measurement. On the other hand, if the interval is too big, it will take the controller a long time to adapt to changing workload conditions. In our experiments we fixed the interval length to 60 seconds. The rationale is that 60 seconds would give the server enough time to stabilize before making further MPL adjustments.

The controller's output is the MPL that the server will use during the next control interval.

B. Auto-tuning Algorithms

The tuning algorithms that we present in this section are based on the Incremental Steps (IS) and Parabola Approximation (PA) algorithms proposed and evaluated using simulation by Heiss and Wagner. We have modified those algorithms and implemented them in SQL Anywhere, allowing us to evaluate the modified algorithms experimentally. Table I introduces some notation that we will be using to describe the algorithms. To clarify the notation, Figure 2 shows a time-line that illustrates how the different values relate to each other. The parameter $n^*(t_i)$ is the number of workers during the control interval $t_{i-1} < t \leq t_i$. The $n^*(t_i)$ value is fixed during a control interval and does not change. $P(t_i)$ is the total number of requests that completed during the control interval $t_{i-1} < t \leq t_i$. The workload concurrency level $n(t_i)$ is the workload concurrency level at time t_i.

Notation	Description
t_i	End of the current control interval and start of the next control interval t_{i+1}.
$n^*(t_i)$	The number of workers the server has available during the control interval $(t_{i-1}, t_i]$. This is the control variable.
$n(t_i)$	The workload concurrency level at time t_i.
$P(t_i)$	The actual measured throughput level of the server at time t_i. The throughput is the number of requests that completed during the previous measurement interval.

TABLE I: Notation used by the tuning algorithms

Another concept that we need to introduce is the throughput curve. A throughput curve is a function that describes the

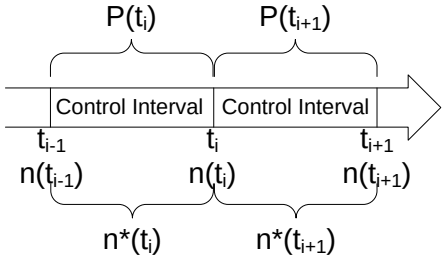


Fig. 2: Control Interval Timeline

relationship between the MPL and the throughput of the server. On a throughput curve, the x-axis is the MPL and is measured by the number of workers. The y-axis is the throughput level. Different workloads have different throughput curves. The shape of a throughput curve depends on many factors including the workload characteristics and the type of hardware or software used.

Our primary goal is to maximize server throughput. Our secondary goal is to minimize the MPL. The secondary goal is included because in some workloads it is possible that the throughput is not very sensitive to the MPL. Having a large number of workers wastes server resources that could have been otherwise used by other components in the database server.

Both of the auto-tuning algorithms monitor the workload concurrency level $n(t_i)$ and behave the same way in case the workload concurrency level is less than the number of workers $n^*(t_i)$. In this case, both algorithms reduce the number of workers gradually to match the measured concurrency level. This is because any workers above and beyond the number needed to accommodate the workload concurrency level will be idle and will not provide any benefit.

1) Hill Climbing: The original IS algorithm proposed by Heiss and Wagner tries to increase the MPL by a fixed amount in each control interval until performance starts to drop. It then turns around and decreases the MPL until performance starts to drop, then turn around, and so on. By doing so, the algorithm attempts to climb to the top of the throughput curve.

Our Hill Climbing algorithm is similar, except that when increasing the concurrency level, it stops if the marginal throughput gain from the additional worker is not enough. The reason for this is that we have observed that throughput curves may have large flat regions on the top and we prefer to stay at low-MPL end of those flat regions rather than wandering within them as the original IS algorithm would do.

Our Hill Climbing algorithm keeps on following the throughput curve as long as the percentage of change in throughput is at least C times the percentage change in the number of workers. On the other hand, if decreasing the number of workers increases performance, then we keep doing so until the performance starts to degrade. The difference between the upward direction and downward direction is due to the fact that this algorithm has built-in bias towards the downward direction. This bias is included to accomplish our second goal of keeping the number of workers as low as possible as well as preventing the controller from going astray in cases where the throughput curve flattens out. If the algorithm does not see any gains in throughput in either the forward or backward direction, it will backtrack to the previous MPL setting.

The second parameter that this algorithm depends on is the step size Δ. The step size controls how fast the algorithm converges.

The algorithm can be described by the following formula:

$$n^*(t_{i+1}) := \begin{cases} n^*(t_i) + \Delta & \text{if } n^*(t_i) \geq n^*(t_{i-1}) \text{ and} \\ & \frac{P(t_i)-P(t_{i-1})}{P(t_{i-1})} \geq C \frac{n^*(t_i)-n^*(t_{i-1})}{n^*(t_{i-1})} \\ n^*(t_i) - \Delta & \text{if } n^*(t_i) < n^*(t_{i-1}) \text{ and} \\ & P(t_i) > P(t_{i-1}) \\ n^*(t_{i-1}) & otherwise \end{cases}$$

where the parameter C takes values from 0 to 1.

This algorithm basically tries to build on the fact that each additional worker added to the worker pool is going to have a smaller benefit (i.e. requests completed) compared to the previous worker added. This happens because every additional thread added is going to consume additional memory (for stack and other thread local storage data) and additional CPU overhead (context switching time and CPU cache pressure). In [16] we studied how those parameters effect the performance of the algorithm.

2) Global Parabola Approximation: This algorithm attempts to model the throughput curve as a parabolic function. We chose a parabola as a model for several reasons. First, a parabola can have a concave downward shape similar to the throughput function found in many servers. Second, the parabola is easy to use and easy to compute. Third, although the real throughput function might have a tail that flattens out, the first part of the throughput function can be approximated by a parabola and we do not need to model the tail part where the server can have degraded performance levels.

In order to approximate a parabola, we need at least three data points. Two of the data points will be based on two observations of throughput at some MPL. For the third point we will use the origin, hence, the global aspect of the parabola approximation. The equation of the throughput curve as a parabolic function is:

$$P(t_i) \quad = \quad a(n^*(t_i))^2 + b(n^*(t_i)) + c$$

The two points that we will be using are ($n^*(t_i)$,$P(t_i)$) and ($n^*(t_{i-1})$,$P(t_{i-1})$). Using the following equations, we can solve for a, b, and c:

$$a \quad = \quad \frac{P(t_i)n^*(t_{i-1}) - P(t_{i-1})n^*(t_i)}{n^*(t_i)n^*(t_{i-1})(n^*(t_i) - n^*(t_{i-1}))}$$

$$b \quad = \quad \frac{P(t_{i-1}) - a(n^*(t_{i-1}))^2}{n^*(t_{i-1})}$$

$$c \quad = \quad 0$$

Once we have the values of a, b, and c, the algorithm tries to move to the n^* that maximizes throughput. This would be the point where the slope is 0.

$$n^*(t_{i+1}) = \frac{-b}{2a}$$

Because of possible noise in the measurements, it is possible that a could be positive. A positive a coefficient models a parabola that is concave up and, hence, does not model a realistic throughput curve. Figure 3 illustrates how noise at MPL 180 generated a concave up parabola. In this example, the controller would suggest moving to point 45. This is clearly a bad step from 180.

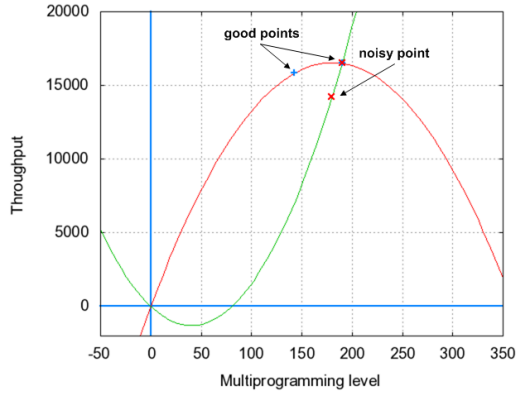


Fig. 3: A concave up and a concave down parabola

If the computed parabolic function is upward opening then we ignore the collected data and just move forward (or backward) by a random amount between 0 and Δ. We keep on alternating between moving forward or backward every time we hit this condition until we start getting more appropriate parabolic functions with positive a coefficient.

Noise in the measurement data can also cause the controller to suggest a new value that is too large. To avoid extreme jumps in the MPL, we use a parameter Γ to cap the step size. The cap is expressed as a percentage of the current number of workers. For example, a Γ value of 0.3 would cap the step size to 30% of the current number of workers. In [16], we discuss in detail on how we choose the values of this parameter and its affect on the performance of the algorithm.

The algorithm actions can be summarized by the following function:

$$n^*(t_{i+1}) := \begin{cases} min(\frac{-b}{2a}, (\Gamma+1)n^*(t_i)) & \text{if } a < 0 \text{ and } \frac{-b}{2a} \neq n^*(t_i) \\ n^*(t_i) \pm rand(0, \Delta) & otherwise \end{cases}$$

IV. Workloads and Evaluation Methodology

We have implemented the Hill Climbing and Global Parabola approximation algorithms in SQL Anywhere. For our experiments, we derived three workloads based on the TPC-C [15] benchmark. The TPC-C workload simulates the activities of an online transaction processing (OLTP) application. There are five types of transactions in the TPC-C workload: New Order, Payment, Order-status, Delivery, and Stock-level. The performance metric for the TPC-C benchmark is the number of New Order transactions that complete within the 90th percentile of a target transaction response times. However, for our experiments we used the total number of server requests completed per control interval as our performance metric, as this is what is optimized by the tuning algorithms. This metric includes operations resulting from all types of transactions, not just New Order transactions. We used two different transactions mixes: TPCC1 and TPCC2. The TPCC1 transaction mix is the same as the standard TPC-C benchmark. TPCC2 uses a slightly modified mix. Table II shows the transaction mix of the two configurations.

Transaction type	TPCC1	TPCC2
New Order	45%	25%
Payment	43%	63%
Order-status	4%	4%
Delivery	4%	4%
Stock-level	4%	4%

TABLE II: Transaction mix TPCC1 and TPCC2

Unlike TPCC1, the TPCC2 workload has fewer I/O requirements as it processes fewer New Order transactions. The TPCC2 workload might not be realistic (more Payment transactions than New Orders) but we use it here because it will have a different throughput curve than TPCC1.

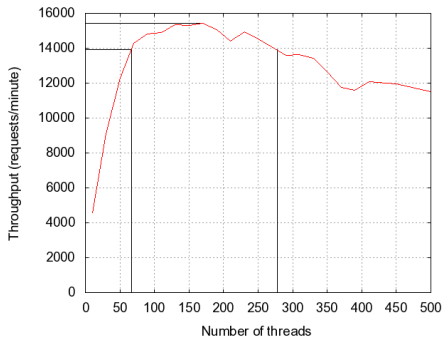
In addition to varying the transaction mix, we varied the database server buffer pool size. We used two buffer pool configurations: SMALLMEM and BIGMEM. In the SMALLMEM configuration, we configured the server buffer pool size to 500MB. In the BIGMEM, it was configured to 1GB. The TPC-C database that we used in our workloads was a 150 warehouse database. Its size is 13.8GB. For the client application, we used 10 terminals per warehouse. The server machine is a 32-bit dual 1.80GHz Intel Xeon(Prestonia) processor machine running Windows 2003 Enterprise Server.

With different transaction mixes and different buffer pool configurations, we now have three different workload configurations based on the TPC-C benchmark:

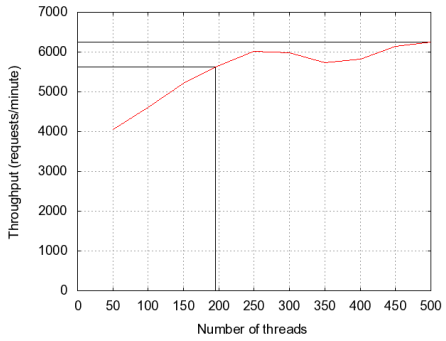
- TPCC1-BIGMEM
- TPCC1-SMALLMEM
- TPCC2-BIGMEM

By experimenting with each workload separately, we generated a throughput curve for each workload. Figure 4 shows the throughput curves for two of the TPC-C workloads. The figure also shows the range of MPL settings for which it is possible to achieve at least 90% of the maximum possible throughput.

We performed two sets of experiments. In the first set, the database server was configured to use one of the auto-tuning algorithms and a benchmark was performed using each of the three different workloads. For each run, the database was reset and the benchmark was restarted. In this set of experiments we measured how quickly and how closely the automatic

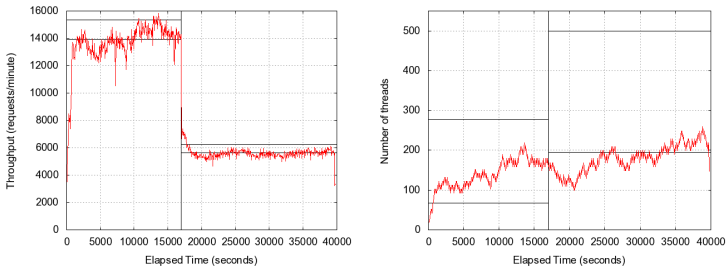


(a) TPCC1-BIGMEM

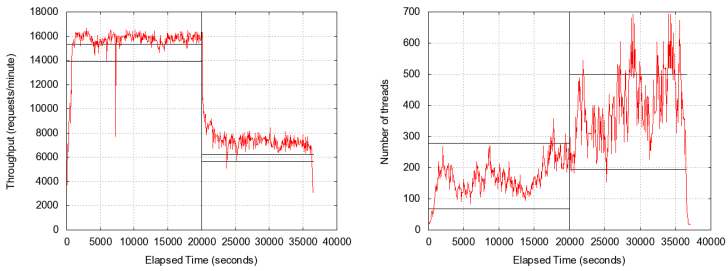


(b) TPCC1-SMALLMEM

Fig. 4: Throughput curves of TPCC1-BIGMEM and TPCC1-SMALLMEM



(a) Throughput (b) MPL



(c) Throughput (d) MPL

Fig. 5: Changing workloads experiment from TPCC1-BIGMEM to TPCC1-SMALLMEM. 5a and 5b show the Hill Climbing algorithm. 5c and 5d show the Global Parabola approximation algorithm

tuning algorithms were able to reach the optimal MPL for each workload.

In the second set of experiments, we ran the server using each of the auto-tuning algorithms but switched from one workload to another half-way through the experiment. We experimented with the following workload switching combinations:

- TPCC1-BIGMEM to TPCC1-SMALLMEM
- TPCC1-SMALLMEM to TPCC1-BIGMEM
- TPCC1-BIGMEM to TPCC2-BIGMEM
- TPCC2-BIGMEM to TPCC1-BIGMEM

The purpose of the changing workload experiments was to understand how well each algorithm can adapt to changes in the workload characteristic.

V. Experimentation Results

Because of space limitations, we will only show the results of one of the changing workload experiments. The graphs in Figure 5 show both throughput and MPL as functions of time. The throughput graphs show the measured server throughput, and the MPL graphs show the MPL setting chosen by the tuning algorithm. In all these graphs there is a vertical line dividing each graph into a left and right section. The vertical line is the point in time at which the workload switch happened. The horizontal lines indicate the regions of optimal values (throughput or MPL) as characterized by the throughput curve of each workload.

Figure 5 shows how the two algorithms behaved when the workload switched from TPCC1-BIGMEM to TPCC1-SMALLMEM. In this specific experiment, we have set the Hill climbing algorithm parameters as follows: $C = 0.4$ and $\Delta = 10$. For the Global Parabola approximation, we have set $\Gamma = 0.333$ and $\Delta = 10$. Across a variety of experiments like those shown in Figure 5, we found [16] that both algorithms were able to react appropriately to workload changes. With our selected parameter settings, the Hill Climbing algorithm is smoother than the Global Parabola technique, though it may take longer to approach near-optimal MPL settings. It is possible to make the Hill Climbing algorithm somewhat more aggressive by increasing the Δ parameter. However, Δ cannot be increased too much without affecting the ability of the Hill Climbing algorithm to fine tune the MPL setting, as it never makes steps smaller than Δ.

In contrast, the Global Parabola Approximation technique is able to make both large steps and small steps, and is therefore able to react relatively quickly to workload changes. The drawback of this this approach is that it is very sensitive to normal, minor workload fluctuations. This sensitivity accounts for the relatively large throughput and MPL fluctuations that can be seen in Figure 5.

VI. Practical Experience

Given the results of our experiments with the Hill Climbing and the Global Parabola approximation algorithms, we have decided to use a hybrid approach in SQL Anywhere 12. The hybrid approach consists of doing automatic tuning using the

Hill Climbing approach for a number of control intervals followed by a switch to the Parabola Approximation approach for one control interval. We then go back and repeat this cycle. During the Hill Climbing control period we collect data points. We then fit the collected data to a parabola using the least squares approximation method. We found using this method allows us to generate better parabola approximation functions. In some cases the parabola approximation may produce a concave up curve. If this happens, then we ignore the decision by the parabola approximation algorithm and use the Hill Climbing algorithm to decide on the next choice of the MPL. We have also modified the Hill Climbing approach to use a variable step size Δ instead of the fixed value. The step size is computed by solving for $n^*(t_i)$ that achieves our C value. We have limited the step size Δ to Γ of the current number of threads. Another change we made was to reduce the control interval from 60 seconds to 30 seconds.

By combining the two algorithms, we are able to make adjustments to the MPL setting without large oscillations. In addition, we are able to reduce convergence times by periodically using the parabola approximation algorithm and using variable step sizes for the Hill Climbing algorithm.

With the hybrid approach we ran a TPC-C benchmark but allowed the server to pick the MPL automatically. The hybrid approach was able to reach 90-95% of the optimal tpmC compared to a hand tuned server. In another internal benchmark that was mostly CPU bound, the hybrid approach was able to improve throughput by 32% compared to fixing the MPL. The tuning algorithm achieved this improvement by reducing the MPL setting to the minimum possible value, which is equal to the number of cores on the system.

One limitation that we have observed is that this form of automatic tuning is not effective for workloads that have a pattern of bursty requests followed by long periods of idle time. It is also not effective for workloads that consist of requests that take a very long time to complete (e.g. creating an index). For such workloads the algorithms took too long to react and possibly limited the concurrency level of the server or reacted in the wrong direction. It was more efficient to fix the MPL setting for these types of workloads. Adjusting the MPL dynamically is more beneficial in *busy* servers where there is a continuous stream of requests. One avenue of further research is to be able to detect and address workloads that have those patterns of requests.

Another limitation of these algorithms is how well the algorithms' parameters are chosen. Different values can have completely different outcomes. In our implementation in SQL Anywhere 12 we have fixed the parameter C to 0.2 for the Hill Climbing algorithm. We have reduced the C value compared to our experiments to make the Hill Climbing algorithm a little more aggressive in climbing the throughput curve.

VII. Conclusions

In this paper we presented a controller that can adjust the MPL of a database server by monitoring throughput level at the server. We presented a Hill Climbing algorithm and a Global Parabola approximation algorithm. We have shown that each algorithm has its own strengths and weaknesses. Nonetheless, these algorithms were able to adjust the MPL effectively. In implementing the automatic tuning of MPL in SQL Anywhere 12 we have used a hybrid approach. In this approach we combined both algorithms in order to gain the benefits of both. The hybrid approach shows significant improvement over each of the algorithms separately.

Acknowledgment

We would like to especially thank Mark Culp for his guidance and various discussions while implementing these algorithms in SQL Anywhere 12.

References

[1] S. Chaudhuri and G. Weikum, "Rethinking database system architecture: Towards a self-tuning RISC-style database system," in *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 1–10.

[2] G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback, "Self-tuning database technology and information services: from wishful thinking to viable engineering." in *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, 2002, pp. 20–31.

[3] G. Weikum, C. Hasse, A. Mönkeberg, and P. Zabback, "The COMFORT automatic tuning project," *Information Systems*, vol. 19, no. 5, pp. 381–432, 1994.

[4] I. T. Bowman, P. Bumbulis, D. Farrar, A. K. Goel, B. Lucier, A. Nica, G. N. Paulley, J. Smirnios, and M. Young-Lai, "SQL Anywhere: A holistic approach to database self-management," in *ICDE Workshops*. IEEE Computer Society, 2007, pp. 414–423.

[5] J. Teng, "Goal-oriented dynamic buffer pool management for data base systems," in *ICECCS '95: Proceedings of the 1st International Conference on Engineering of Complex Computer Systems*. Washington, DC, USA: IEEE Computer Society, 1995, p. 191.

[6] M. Greenwald, "Practical algorithms for self scaling histograms or better than average data collection," *Perform. Eval.*, vol. 27/28, no. 4, pp. 19–40, 1996.

[7] "Microsoft SQL Server: How to determine proper SQL Server configuration settings," http://support.microsoft.com/kb/319942.

[8] iAnywhere Solutions, *SQL Anywhere Server Database Administration: Setting the database server's multiprogramming level*.

[9] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman, "How to determine a good multi-programming level for external scheduling," in *Proceedings of the 22nd International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, p. 60.

[10] S. Harizopoulos, "Staged database systems," Ph.D. dissertation, Carnegie Mellon University, 2005.

[11] J. Ousterhout, "Why threads are a bad idea (for most purposes)," *Presentation given at the 1996 Usenix Annual Technical Conference, January*, 1996.

[12] S. Chen and I. Gorton, "A predictive performance model to evaluate the contention cost in application servers," in *APSEC '02: Proceedings of the Ninth Asia-Pacific Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2002, p. 435.

[13] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," in *WWW '04: Proceedings of the 13th international conference on World Wide Web*. New York, NY, USA: ACM Press, 2004, pp. 287–296.

[14] H.-U. Heiss and R. Wagner, "Adaptive load control in transaction processing systems," in *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991, pp. 47–54.

[15] Transaction Processing Performance Council, "TPC Benchmark C, Standard Specification," http://www.tpc.org/tpcc.

[16] M. Abouzour, "Automatically tuning database server multiprogramming level," Master's thesis, University of Waterloo, 2007.